In *Scientific Programming* **8** (2000), pp. 183–193.
Corrected reprint of article first appearing in *Proceedings SC 2000* (Supercomputing).

Scalable Algorithms for Adaptive Statistical Designs

Robert Oehmke Janis Hardwick Quentin F. Stout

University of Michigan
Ann Arbor, Michigan 48109 USA

Abstract

We present a scalable, high-performance solution to multidimensional recurrences that arise in adaptive statistical designs. Adaptive designs are an important class of learning algorithms for a stochastic environment, and we focus on the problem of optimally assigning patients to treatments in clinical trials. While adaptive designs have significant ethical and cost advantages, they are rarely utilized because of the complexity of optimizing and analyzing them. Computational challenges include massive memory requirements, few calculations per memory access, and multiply-nested loops with dynamic indices. We analyze the effects of various parallelization options, and while standard approaches do not work well, with effort an efficient, highly scalable program can be developed. This allows us to solve problems thousands of times more complex than those solved previously, which helps make adaptive designs practical. Further, our work applies to many other problems involving neighbor recurrences, such as generalized string matching.

Keywords: dynamic programming, computational learning theory, bandit models, message-passing, dynamic domain decomposition, memory-intensive computing, load balancing, sequential analysis, performance analysis, experimental algorithms

1 Introduction

Standard statistical designs define all sampling decisions in advance. In particular, in a clinical trial, the number of patients that will receive each treatment is decided before the trial begins. In contrast, adaptive designs use accruing information to adjust the decisions dynamically. For example, if in the midst of a trial it has been observed that one treatment is performing better than the others, then more patients may be assigned to the apparently better treatment. Thus adaptive designs can provide significant ethical benefits, and in industrial settings can have significant cost and time advantages [6]. However, adaptive designs are rarely used, largely because they are far more difficult to analyze. Analytical solutions are impossible in all but the most trivial cases, and computational approaches are often considered infeasible.

We are developing new algorithms, and optimized implementations, to solve adaptive design problems. Here we primarily report on one case, optimizing an n-stage trial with three treatment options having Bernoulli outcomes. This problem is translated into a 6-dimensional dynamic program for which we developed a highly scalable solution, allowing us to create designs of useful size.

This dynamic programming problem is a neighbor recurrence where the value at a given location is determined by the values at a stencil of other locations that are "near" in the parameter space. Neighbor recurrences are quite common, such as the Fibonacci sequence $F(n) = F(n-1) + F(n-2)$ or in the use of dynamic programming to solve optimization problems such as the alignment of gene or protein structures in bioinformatics. They also occur in backwards induction and path induction [7].

Unfortunately, the computational complexity of such recurrences grows exponentially in the dimension. This "curse of dimensionality" often makes exact solutions infeasible, and thus approximations are used and the solution quality is reduced. Since there is considerable interest in solving such computationally formidable recurrences, parallel computing is a natural approach. It is generally, but mistakenly, felt that the regularity of neighbor recurrences implies that parallelization is straightforward. While high efficiency can be attained, it requires considerable effort. Major difficulties include:

- Time and space grow rapidly with the input size, so intensive efforts are needed to obtain a useful increase in problem size.
- The time/space ratio is low, making RAM the limiting factor.
- There are few calculations per memory access.
- The nested loops have dynamic index dependencies.

Performance is further exacerbated by the interaction of these aspects. Table 9 shows, for example, the dramatic limitations imposed by space constraints and imperfect load balance caused by the loop structure.

Section 1.1 details the primary example problem, and Section 1.2 discusses prior work. Section 2 shows a natural serial implementation and space reductions. Section 3 discusses an initial, natural, distributed memory parallelization and its inadequacies. Section 4 develops a scalable parallelization, with timing analyses in Section 4.2 and a discussion of a performance degradation in Section 4.3. Section 5 develops and analyzes shared memory parallelization. Section 6 discusses the distributed memory parallelization of a related but more difficult problem involving delayed responses, and highlights new complications that arise. Section 7 provides a final discussion.

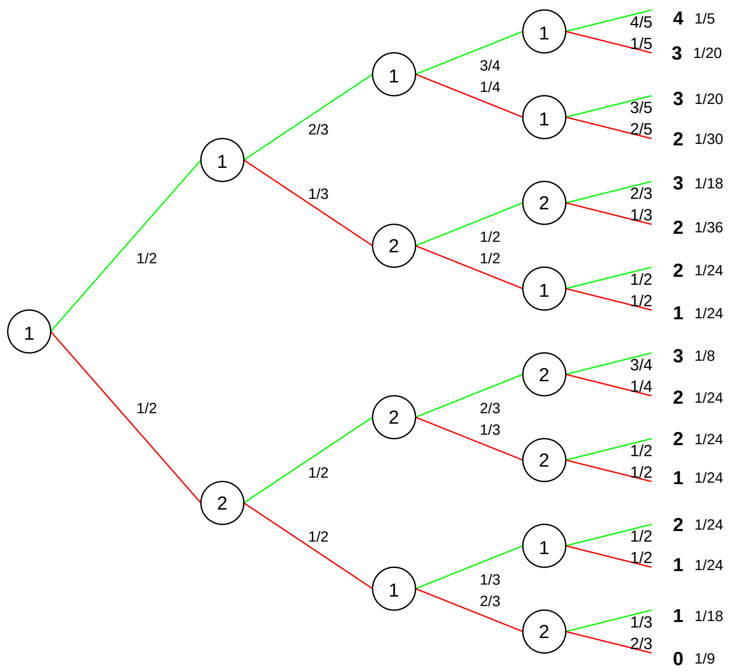
The distributed memory results were obtained using MPI on an IBM SP2, where each processor is an 160 MHz POWER2 Super Chip (P2SC) processor with 1 GB of RAM and 1 GB additional virtual memory. The shared memory results were obtained on a 16 processor SGI Origin with 12 GB RAM, where each processor is a 250 MHz MIPS R10000. Throughout, all times are elapsed wall-clock time measured in seconds. Rerunning the same problem showed very little timing variation, so we merely report average time (see, however, Section 4.3).

1.1 Multi-Arm Allocation

Sequentially allocating patients to treatment options so as to optimize their outcomes in a clinical trial can be modeled as a Bayesian *bandit problem* [2]. Such models are important in stochastic optimization as well as in decision and learning theory. In a k-arm bandit problem one can sample from any of k independent arms (populations) at each stage of the experiment. (Here, "arm" = "treatment option".) Associated with each arm is a prior distribution on the unknown outcome or "reward" function. After sampling from an arm (e.g., allocating a patient to a treatment) one observes the outcome and updates the information (prior) for that arm. The goal is to *determine how best to utilize accruing information to optimize the total outcome for the experiment*. In this case, the outcome functions are independent Bernoulli random variables, resulting in "success" or "failure", and the goal is to maximize the number of successes.

At each stage, $m = 0, ..., n-1$ of an experiment of length n, an arm is selected and the response is observed. At stage m, let (s_i, f_i) represent the number of successes and failures from arm i. Then the *state* $(s_1, f_1, \ldots, s_k, f_k)$, is a vector of sufficient statistics. Optimal solutions can be obtained via dynamic programming, but the time and space have the formidable growth rate of $\Theta(n^{2k}/(2k-1)!)$. We concentrate on the 3-arm version, which has $\Theta(n^6)$ complexity.

Figure 1 illustrates a simple 2-arm bandit design, where each arm has a uniform prior on its rate of success, for $n = 4$. Non-adaptive designs would only average 2 successes, while by adapting the optimal design achieves 2.27. The advantages of adaptation become more pronounced the longer the trial is and the



Node: arm sampled; Right cols: successes achieved and prob. reaching that outcome.
Upward line: success; Downward line: failure; Line label: prob. of outcome.

Figure 1: A 2-arm bandit, with $n = 4$ and uniform priors on each arm.

---

more arms there are. For example, with $n = 100$ and uniform priors on each arm, non-adaptive allocation will average 50 successes no matter how many arms there are. However, the optimal 2-arm bandit will average 65 successes, and the 3-arm bandit averages 72.

1.2 Previous Work

The 3-arm problem had never previously been solved exactly because it was considered infeasible. Indicating frustration with the far easier *2-arm* bandit problem, researchers have commented: "In theory the optimal strategies can always be found by dynamic programming but the computation required is prohibitive" [20], and "the space and time requirements for this computation grow at a rate proportional to n^4 making it impractical to compute the decision even for moderate values of say $n \geq 50$" [9]. Previously, the largest exact 2-arm bandit solution utilized a Cray 2 supercomputer to solve n=200 [3]. Here, we solve a problem *2,000* times harder, namely the 3-arm bandit with $n = 200$.

There appears to be no previous work on the parallel solution of bandit problems, but there has been considerable work on the parallel solution of similar recurrences. Most of this concentrates on theoretical algorithms where the number of processors scales far faster than the input size [8, 14, 15, 12, 16], or special purpose systems are created [17, 10]. Others [11, 19] look at dynamic programming difficulties when the subproblems are not as well understood.

Algorithm 1 Serial Algorithm for Determining Optimal Adaptive 3-Arm Allocation

{$\widehat{s_i}$, $\widehat{f_i}$: one success, failure on arm i }
{si, fi: number of successes, failures arm i }
{m: number of observations so far }
{n: total number of observations }
{$|\sigma|$: number of observations at state σ }
{V: the function being optimized, where V(0) is the answer }
{pi(si,fi): prob of success on arm i, if si successes and fi failures have been observed }

for all states σ with $|\sigma|$=n **do** {i.e. for all terminal states}
  V(σ)=number of successes in σ

for m=n-1 downto 0 **do** {compute for all states of size m}
  for s3=0 to m **do**
    for f3=0 to m-s3 **do**
      for s2=0 to m-s3-f3 **do**
        for f2=0 to m-s3-f3-s2 **do**
          for s1=0 to m-s3-f3-s2-f2 **do**
            f1 = m-s3-f3-s2-f2-s1
            σ = $\langle$s1,f1,s2,f2,s3,f3$\rangle$
            V(σ) = max{(p1(s1,f1) $\cdot$ V($\sigma + \widehat{s_1}$) + (1-p1(s1,f1)) $\cdot$ V($\sigma + \widehat{f_1}$)) ,
              (p2(s2,f2) $\cdot$ V($\sigma + \widehat{s_2}$) + (1-p2(s2,f2)) $\cdot$ V($\sigma + \widehat{f_2}$)) ,
              (p3(s3,f3) $\cdot$ V($\sigma + \widehat{s_3}$) + (1-p3(s3,f3)) $\cdot$ V($\sigma + \widehat{f_3}$)) }

2 Serial Implementation

The goal of a bandit problem with dichotomous responses is to determine, at each state, which arm should be selected so as to maximize the expected number of successes over the course of the experiment. To solve this via standard dynamic programming (Algorithm 1), first the values of each terminal state (those with n observations) are computed. Then, the optimal solution is found for all states with m observations based on the optimal solutions for all states with $m + 1$ observations, for $m = n - 1$ down to 0.

The neighbor recurrence at the heart of this algorithm is in the center of the loops. For our purposes, the specific function used to combine values is less important than the indices of the values being referenced, since they determine the memory accesses and communication required. Note that we have a stencil of dependencies, whereby the value at state σ depends only on the neighbor values at $\sigma + \widehat{s_1}$, $\sigma + \widehat{f_1}$, $\sigma + \widehat{s_2}$, $\sigma + \widehat{f_2}$, $\sigma + \widehat{s_3}$, and $\sigma + \widehat{f_3}$, where $\widehat{s_i}$ and $\widehat{f_i}$ denote a single additional success or failure, respectively, on arm i. With minor changes to this equation (and no change in the dependencies), the same program can also perform *backward induction* to evaluate the expected number of successes for an arbitrary 3-arm design, which allows one to evaluate suboptimal designs which may have other desirable characteristics. Further, trivial changes allow evaluation of far more general objective functions.

Note that the recurrences involve extensive memory accesses, with little computation per access. There are $\binom{n+6}{6} = \Theta(n^6)$ states, and the time and space complexities are also $\Theta(n^6)$.

n	first	collapsed	naive comp	best
10	.009	.004	.082	.004
20	.18	.1	3.2	.092
30		1.4	35	.71
40		4.1	186	3.9
50		15	689	13
60			2024	35
70			5047	86
80			11242	185
90			22756	362
100			42900	659
110				1225
120				1922
130				34961
max n	27	54	100	135
limitation	memory	memory	time	time
prog len	193	193	282	419

max n: Maximum problem solvable with 1 GB and time $\leq$ 64,400 sec. (18 hr.)

Table 1: Serial versions, time (sec.) to solve problem of size n.

2.1 Space Optimizations

The first space reduction results from the observation that values of `V` for a given m depend only on the values for $m+1$, so only the states corresponding to these two stages need to be kept simultaneously. This reduces the working space to $\Theta(n^5)$, and by properly arranging the order of the calculations, the space can be reduced to only that required for one stage's worth of states, i.e., we gain another factor of 2. This corresponds to the *collapsed* column in Table 1. In this table, *max* n shows the maximum problem solvable by a 1 GB RAM machine with a time limit of 18 hours, *limitation* shows which limit was reached, and *prog len* is the size of the version in lines of source code. Note that the collapsed version allows us to solve problems substantially larger, and also results in a slight speedup.

The next space reduction results from the fact that, due to the constraint `s3+f3+s2+f2+s1+f1` $\leq n$, only a corner (approximately 1/5! = 1/120 of the total) of the 5-dimensional `V` array is used. To take advantage of this, the 5-dimensional `V` array is mapped into a linear array. Unfortunately, this mapping also requires all array references to be translated from the original five indices into their position in the linear array. From a software engineering viewpoint, the best way to implement this translation is to use a function which takes as input the five indices and yields their position in the array. Unfortunately, this is extremely costly as the translation function is a complicated 5^{th} degree polynomial which must be evaluated for every array access. This version, the *naive comp* in Table 1, can solve larger problems, but is significantly slower than the *collapsed* version. For the *best* version, we broke the translation function into a series of offset functions, where each offset function corresponds to a given nested loop level. An offset function only needs to be recalculated before its corresponding loop is entered, and the more expensive offset functions correspond to the outermost loops. This method dramatically reduces the translation cost down to a usable level, but greatly increases program complexity, as is shown by the increase in *prog len*.

The simplified Algorithm 1 ignores the fact that in order to utilize the design, one needs to record the arm selected at each state. This is typical with dynamic programming. Unfortunately these values cannot be overwritten and the storage required is $\Theta(n^6)$. Fortunately, this too involves only values in one corner, allowing a reduction by a factor of 1/6! = 1/720. These values are stored on disk and do not reduce the amount of memory available for calculation. Using run-length encoding would reduce this to $\Theta(n^5)$, but so far this has not been necessary.

3 Initial Parallel Algorithm

To parallelize the recurrence, we first address load balancing. In the initial parallelization the natural approach of dividing the work among the processors was taken. The outermost `m` loop behaves very much like *time* and cannot be parallelized, so instead one parallelizes the second outermost loop, `s3`. At stage `m`, processor $\mathcal{P}_j$ is assigned the task of computing all values where `s3` was in the range `start_s3(j,m) ...end_s3(j,m)`.

Determining the range of `s3` values assigned to each processor is nontrivial, because the number of states corresponding to a given value of `s3` grows as $(m - s3)^4$. Thus, simply assigning all processors an equal number of `s3` values would result in massive load imbalance and poor scaling. We evaluated two solutions to this problem. Optimal `s3` partitioning is itself a small dynamic programming problem which takes time and space $\Theta(mp)$. However, it was easy to develop a fast $\Theta(m)$ greedy heuristic which was nearly optimal, and it is this heuristic which was used in the program.

3.1 Communication

The communication needed can be divided into *array redistribution* and *external neighbor acquisition*. Array redistribution occurs because, as the calculation proceeds, the number of states shrinks. To maintain load-balance, the `s3` range owned by a processor changes over time. At stage `m`, processor $\mathcal{P}_j$ needs the states with `s3` values in the range `start_s3(j,m)...start_s3(j,m+1)-1` from $\mathcal{P}_{j-1}$. Redistribution includes the cost of moving the states currently on the processor to create space for these new states.

External neighbor acquisition occurs because the calculations for a state may depend on its neighbors in other processors. To calculate states with `s3=end_s3(j,m)` during stage `m`, $\mathcal{P}_j$ needs to obtain a copy of the states with `s3=end_s3(j,m)+1` from $\mathcal{P}_{j+1}$. Note that external neighbor acquisition negates round-robin or self-scheduling approaches to load-balancing the `s3` loops, as this would result in a dramatic increase in the communication requirements. This does not necessarily hold for shared memory systems, however, as can be seen from the OpenMP version in Section 5. Shared memory computers are able to utilize these approaches because their much faster communication systems reduce the latency down to a managable level.

4 Scalable Parallel Algorithm

The initial load-balancing approach is simple to implement and debug because it makes minimal changes to the serial version. Unfortunately, it has imperfect load and working space balancing, which severely limits scalability (see Table 2) and the size of problem solvable (see Table 9).

For a more scalable version (Algorithm 2), instead of partitioning the states using the coarse granularity of the `s3` values, we partition them as finely as possible. However, this leads to numerous difficulties. The first is that a processor's `V` array can now start or end at arbitrary values of `s3, f3, s2, f2, s1,` and `f1`, so one can no longer use a simple set of nested loops to iterate between the start and end value. Our

Algorithm 2 Scalable Parallel Algorithm

```
{P_j: processor j}
{start_σ(j,m), end_σ(j,m): range of σ values assigned to P_j for this m value,
 with start_σ(j+1,m)=end_σ(j,m)+1 }

{For all processors P_j simultaneously, do}

for σ=start_σ(j,n) to end_σ(j,n) do {i.e. for all terminal states}
   V(σ)=number of failures in σ

for m=n-1 downto 0 do {compute for all states of size m}
   for σ=start_σ(j,m) to end_σ(j,m) do
      determine s1, f1, s2, f2, s3, f3 from σ
      compute V as before

   {Array redistribution}
   Send needed V values to other processors
   Receive V values from other processors

   {External data acquisition}
   Send needed V values to other processors
   Receive V values from other processors
```

first attempt to solve this problem had nested if-statements within the innermost loop, where the execution rarely went deep within the nest. While logically efficient, this turned out to be quite slow because it was too complex for the compiler to optimize. A solution that the compiler was able to cope with was to use a set of nested loops with if-statements in front of each loop so that it starts and stops appropriately. This solution was almost as fast as the original serial nested loops.

Another difficulty was that the offset calculations are not uniformly distributed along the range of the `V` array, and this leads to a noticeable load imbalance. Storing the results of the offset equations in arrays significantly decreases the cost of each offset calculation and reduces the load imbalance to a more acceptable level. However, there is still some slight load imbalance that could be addressed by including the cost of these array lookups in the load balancing.

4.1 Communication

The move to perfect division of the `V` array also caused complications in the communication portion of the program. The main complication was that data needed for either external or redistribution aspects was no longer necessarily located on adjacent processors. This resulted in a considerable increase in the complexity of the communication portions of the program.

Our initial version of the communication functions used a natural strategy when space is a concern: each processor sent the data it needed to send, shifted its remaining internal data, and then received the data sent to it. Blocking sends were used to insure that there was space to receive the messages. Unfortunately, this serialized the communication, because the only processor initially ready to receive was the one holding the end of the array, i.e., the only processor which does not redistribute to any other processor. The next processor able to receive was the second from the end, because it sent only to the end processor, and so on.

p	efficiency $e(p)$	
	initial	scalable
1	1.00	1.00
2	.96	.96
4	.93	.94
8	.81	.91
16	.64	.86
32	.48	.81

Table 2: Scaling results, $n = 100$.

version	$t(1)$	$t(8)$	$e(8)$
first scalable	1044	178	.734
improved loops	775	143	.678
offsets in array	766	134	.715
scalable comm	762	106	.903
non-blocking comm	760	104	.913

Table 3: Stepwise improvements in scalable version, $n = 100$, 1 and 8 processors.

				array redist		external
p	calc	file	misc	comm	shift	comm
1	98	1.9	0.1	0.0	0.0	0.0
2	94	1.6	0.9	1.9	1.2	0.4
4	88	1.6	0.1	4.5	2.0	3.8
8	84	1.4	0.2	6.5	2.0	5.9
16	73	1.2	0.7	11.0	2.1	12.0
32	57	1.1	0.0	16.1	1.7	24.1

Table 4: Percentage distribution of time within scalable version, $n = 100$.

The performance of this version was unacceptable. The next version removed the interaction and performed adequately but synchronization costs became more of a problem. To remove these, we switched to non-blocking communication wherever possible. This made communication fairly efficient, however there may still be room for some slight additional improvements.

In general there is a serious conflict between extensive user space requirements and minimizing communication delays. The communication buffers needed to overlap communication and calculation, and to overlap incomplete sends and receives, can be large.

4.2 Scalable Timing Results

Table 2 shows the efficiency, $e(p)$, of the initial and scalable parallel versions as the number of processors p increases. Table 3 shows the effect on timing and scaling of each of the major changes detailed in Section 4, contrasting 1 processor and 8 processor versions, where $t(p)$ is the time. Note that the improvements reduced the serial time, and increased the parallel efficiency relative to the reduced serial time.

Table 4 contains the percentage of the total running time taken by different parts of the scalable program as the number of processors increases. *Calc* is the percentage of time taken by the dynamic programming calculations, *file* is the cost of writing the decisions to disk, and *misc* is the part of the time not attributed elsewhere. Under *array redist*, we show the cost of shifting data among the processors to maintain load-balance, where *comm* is the cost of calculating the redistribution and communicating the data between the processors, and *shift* is the cost of moving the data on the processor. Below *external comm* is the cost of getting neighbor states from other processors, including the cost of determining which processor has the

p	$t(p)$
16	10463
32	1965

Table 5: Timing results, $n = 200$, scalable version.

	p	1	2	4	8	16	32
old	$t(p)$	760	396	203	104	55	30
	$e(p)$	1.00	.96	.94	.91	.86	.81
new	$t(p)$	854	448	237	131	73	42
	$e(p)$	1.00	.95	.90	.81	.72	.63

Table 6: Comparison of scalable program efficiencies from old to new system, n=100

data and where to put it on the current processor, and the cost of communicating the data.

Table 5 presents the running times of the scalable program for $n = 200$ for 16 and 32 processors. Note that the speedup is more than a factor of two. This occured because on 16 processors the program must make extensive use of disk-based virtual memory. A similar effect can be seen in Table 1 as n increases from 120 to 130. This illustrates an often overlooked advantage of parallel computers, a bonus increase in speed simply because dividing a problem among more processors allows it to run in RAM instead of in virtual memory. However, this can be successful only if the parallelization load-balances the memory and computation requirements.

4.3 System Performance Degradation

While generating the timing analyses for this paper we collected data on the SP2 at two different times approximately a year apart. Much to our consternation we discovered that the most recent data showed the program to be running much slower and scaling poorly. For an example of this change see Table 6 where we present the change in time and efficiency in the scalable version from last year to this year. Note that this slowdown occurred with exactly the same code, in fact with the same binaries, used the previous year. The slowdown occured for multiple versions of our programs, and we even have some evidence of it occuring in different applications, although finding other users with precise timing data from the previous year has proved to be quite difficult.

Questioning the SP2 systems staff revealed that during the year the operating system had been upgraded. Further investigation revealed that the new operating system allowed off-processor access to the local disk on each node. We believe it is this change which resulted in our performance degradation. Allowing off-processor jobs to access the local disk causes contention for the disk, for the high performace switch used in interprocessor communication, and for the CPU, all of which can degrade performance for a program running on the node. Further, the more processors being used, the more likely it is that at least one of them is having its disk accessed. If any user node is delayed then the internode communication dependencies quickly insure that all of the nodes are delayed, which degrades scalability. Since our dynamic programming problems have a low calculation to communication ratio, they are quite sensitive to these effects.

This problem illustrates an interesting dilemma of many computing centers. Allowing users to remotely access local disks increases throughput and is more convienient for the users as a whole, although it hurts the individual user trying to extract maximum run-time performace. The question of whether to optimize for throughput or individual performace is a complex one without easy answers, and our results show that decisions may have more extensive impact than expected.

5 Shared Memory Implementations

To measure the performance of the 3-arm bandit code on a shared memory machine we implemented four seperate versions.

The first version, which we call MPI, uses the shared memory implementation of the MPI libraries. Aside from a few changes due to differences in the versions of Fortran on the two machines, this version is identical to the scalable version of the code previously described.

The next version, OpenMP, uses OpenMP directives to implement a shared memory version of the code. This version is very similar to that in Algorithm 1, except for the addition of a second copy of the `V` array. This second copy is necessary because while using a shared memory implemention the same `V` array is shared among all the processors, which may be acting on different sections of it at arbitrary times. This means there is no longer a guarantee that every calculation that uses a state will be finished before the state is overwritten, and thus we need to have a second array to hold the current stage's inputs while the current stage's outputs are being stored. After a stage is completed its output array is copied into the input array for the next stage.

To convert the code, OpenMP parallel-do directives were used around the outmost, `s3`, loop of the dynamic programming setup, and the `s3` loop in the dynamic programming. Both of these loops use OpenMP dynamic scheduling, which means that each processor grabs a user defined chunk size number of iterations, performs them, and then when completed grabs another set. This process continues until all the iterations of the loop have been completed. To compute the chunk size for each stage, we first determine the average amount of work per processor at that stage. The chunk size is then 1 less than the mininum number of iterations whose combined work is greater than the average work. Note that this will not necessarily be the number of iterations divided by the number of processors since the work in each iteration varies dramatically. This type of dynamic scheduling approach is not feasable for the distributed memory version of our code because of the increase in complexity that would result from tracking the location of the states and synchronizing access to them.

The third version of shared memory code, Auto, was generated by using the SGI Fortran autoparallelizer on the serial version of our code. Unfortunately, due to the dependencies inside the `V` array described above, the autoparallelizer was only able to parallelize the innermost, `s1`, loop of the dynamic programming setup.

The final version of shared memory code, Auto+Copy, again used the autoparallelizer, but this time on the double `V` array code described above for OpenMP. The reduction in dependencies allowed it to do slightly better. It parallelized the innermost, `s1`, loops of both the setup and the main body of the dynamic programming.

Table 7 shows the results of our measurements on these four versions. As can be seen, the hand parallelized versions perform far better than those done automatically. In fact, Auto, has almost no discernable increase in speed as the number of processors increases. Auto+Copy does slightly better, but is still far inferior to the others. The winner clearly is OpenMP, which was to be expected as it has far less overhead than MPI. Note, however, that OpenMP's scalability will degrade as the number of processors increases because it cannot allocate less than one `s3` loop per processor. (Because we have only 16 nodes on our SGI Origin, we could not provide numbers for more processsors). Implementing a fully scalable code using OpenMP would be difficult, and in the end would probably result in something similar to the MPI version.

	MPI		OpenMP		Auto		Auto+Copy	
p	$t(p)$	$e(p)$	$t(p)$	$e(p)$	$t(p)$	$e(p)$	$t(p)$	$e(p)$
1	439	1.00	406	1.00	471	1.00	454	1.00
2	290	.76	209	.97	473	.49	419	.54
4	155	.70	113	.90	465	.25	404	.28
8	90	.61	72	.70	473	.13	403	.14
16	73	.38	59	.43	470	.06	397	.07

Table 7: Efficiency of shared memory implementations, n=100

6 Delayed Response Problem

We have also applied our scalable parallelization approach to a more complex problem involving 2 Bernoulli arms, where now there is no assumption that the responses are obtained immediately. Thus new patients may need to be assigned to treatment even though we have not observed the outcomes of all prior assignments. This *delayed response* situation is a significant practical problem, and is often cited as a difficulty when trying to apply adaptive designs [1, 18]. Moreover, like the 3-arm problem, the delayed response problem had never been fully optimized, neither analytically nor computationally, because it was considered intractible.

There are many different models of the delay, appropriate for varying circumstances. Here we assume that the response times for each arm are exponentially distributed, and that patients arrive according to a Poisson process. In this setting, the natural states are of the form $(s_1, f_1, u_1, s_2, f_2, u_2)$, where u_i is the number of patients assigned to treatment i with unknown outcome. As before, we have the condition that $s_1 + f_1 + u_1 + s_2 + f_2 + u_2 \leq n$, which allows compression, and we have exactly the same number of states as in the 3-arm problem of size n. However, a critical difference is that the recurrence for $V(\sigma)$ depends upon $V(\sigma + \widehat{u_1})$, $V(\sigma + \widehat{s_1} - \widehat{u_1})$, $V(\sigma + \widehat{f_1} - \widehat{u_1})$, $V(\sigma + \widehat{u_2})$, $V(\sigma + \widehat{s_2} - \widehat{u_2})$, and $V(\sigma + \widehat{f_2} - \widehat{u_2})$. That is, either a patient is assigned a treatment and the outcome is initially unknown, or we have just observed the outcome of a treatment. See [5] for the detailed form of the recurrence and its derivation.

Figure 2 shows the effect of delay on the number of successes for a simple problem involving uniform priors on the success rates of each arm, with $n = 100$. If no responses were obtained before all patients were allocated then the expected number of successes would be 50, which could be obtained by equal allocation of patients to each arm. If all responses were immediate and the optimal 2-arm bandit design was used then the expected number of successes would increase to 64.92. Under delayed response we would expect fewer successes, but the optimal number obtainable was previously unknown. In the figure, B represents the optimal design for the known delay parameters, and R represents the most commonly suggested adaptive design for this situation, known as *randomized play the winner* (RPW) [21]. Note that the optimal design is significantly better than RPW, and that it tolerates delays quite well.

While the recurrences for the delayed response model again have a stencil of neighbor dependencies, they are more complicated. To go through the calculations systematically, one needs the appropriate notion of "stage", corresponding to m in the 3-arm program. In general, the stage of a state σ should be the maximum path length to the state from the initial state $\mathbf{0}$. Previously, all paths to σ from $\mathbf{0}$ took the same number of steps, which was the sum of the entries. Here again all paths have the same length, but it is $2(s_1 + f_1 + s_2 + f_2) + u_1 + u_2$, i.e., the components do not contribute uniformly. Because all the paths from σ from $\mathbf{0}$ are the same length, states at stage k (i.e., at distance k) depend only on states at stage $k + 1$, which allows one to store only 2 stages at a time. Further, as in the original problem, by carefully analyzing the dependencies and going through the loops in the correct order, this can be reduced down to 1 stage.

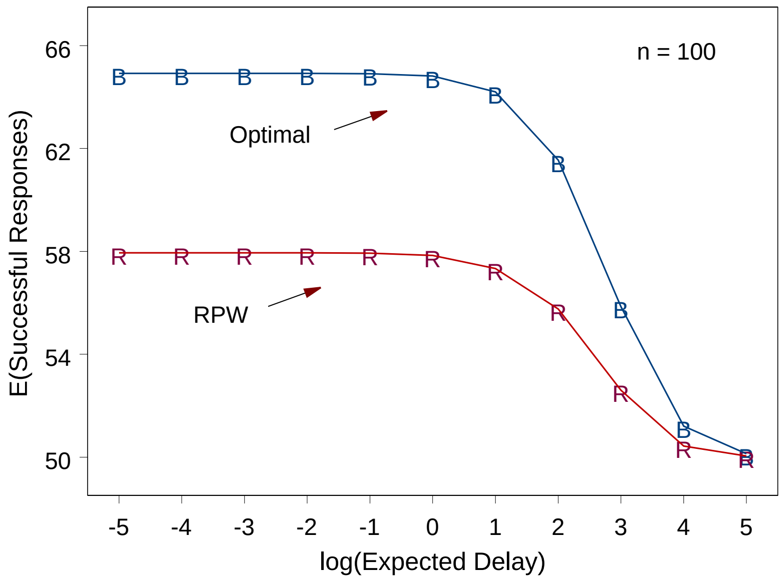


Figure 2: Expected successes vs. delay, expected patient arrival rate = 10^0, uniform priors.

				array redist		external
p	$e(p)$	calc	misc	comm	shift	comm
1	1.00	95.8	0.0	0.0	4.2	0.0
2	.93	89.5	0.0	3.7	3.8	3.0
4	.79	75.7	0.0	12.4	3.6	8.3
8	.67	61.9	0.1	18.4	2.8	16.8
16	.41	41.5	0.2	28.2	2.0	28.1
32	.27	25.8	0.2	31.8	1.2	41.0

Table 8: Analysis of delay program on new system, n=100.

However, there are now $2n$ stages for the outermost loop, as opposed to the n used previously. This has the negative effect of doubling the number of rounds of communication, which significantly reduces the parallel efficiency. It does have a positive effect, however, of slightly reducing the memory requirements since the same number of states are spread over more stages.

The nonuniform roles of the indices make the array compression calculations somewhat more complex, and make it harder to determine the indices of the states depended on. An additional complication comes from the fact that previously, any combination of nonnegative entries having a sum of m was a valid state at stage $m \leq n$. Now, however, there can be a valid stage $m \leq 2n$, and a combination of nonnegative entries having that weighted sum, but the combination does not correspond to a state. For example, if $n = 100$, then $(0, 0, 75, 0, 0, 75)$ is not a valid state, even though it is at stage 150. The reason is that it violates the constraint that $s_1 + f_1 + u_1 + s_2 + f_2 + u_2 \leq n$. Previously this constraint was automatically satisfied, but this is no longer true. This situation complicates the compressed indexing and access processes.

Table 8 contains the timing and scaling analysis of the program, which incorporates all of the features of the most scalable 3-arm program. This was run on the new version of the system, so one would expect

n	uncompressed	initial	scalable
100	100	1	1
200	∞	21	16
300	∞	∞	173

Max problem solvable: uncompressed: 105; initial: 231; scalable: ∞.

Table 9: Min. processors (p) needed to solve problem of size n, using 1 GB per processor.

the performance to be degraded some. However, the drop in efficiency is rather significant, caused by the complex indexing and extra rounds of communication. We assume that another important factor is that the program was only recently developed, and with additional tuning its performance should improve some.

7 Conclusions

There is considerable interest in using adaptive designs in various experiments because they can save lives (human or animal), time, cost, or other resources. For example, for a representative delayed response problem with $n = 100$, uniform priors, and response delay rates 10 times patient arrival rates, simple equal allocation averages 50 successes. The most commonly suggested adaptive technique, randomized play the winner (RPW), achieves only a 14.7% improvement, while the newly obtained optimal solution achieves a 28.4% improvement (see Figure 2). In fact, the optimal solution is nearly as good as the optimal solution for the case where there are no delays. Note that this is also the first exact evaluation of RPW in this setting, using a trivial modification of the optimization program to perform backwards induction.

However, the complexity of adaptive designs has proven to be a major hurdle impeding their use. Our goal is to reduce computational concerns to the point where they are not a key issue in the selection of appropriate designs. This paper has concentrated on the parallel computational aspects of this work, while other papers analyze the statistical and application impact [4, 5, 6].

Unfortunately, the recurrences involved have attributes that make it difficult to achieve high performance and scalability. Space tends to be the limiting factor, and trying to ameliorate this causes overhead and a significant increase in program complexity. As noted in Section 4, increases in program complexity can cause severe performance problems when the compiler is unable to optimize the inner-most loops, and hence one must select alternatives with the compiler's limitations in mind. Space constraints, and low calculation to communication ratios, also complicate the ability to reduce communication latencies. However, by working diligently, it is possible to achieve significant speedups and scalable parallelizations, although this comes at a cost of increased program length and more complex program maintanence. Of course, as was shown in Section 4.3, even highly scalable programs can have their performance degraded in unhelpful environments.

In Table 9 we illustrate the effects of memory limitations on the 3-arm problem, assuming 1 GB per processor. *Uncompressed* refers to a parallel program using load-balancing as in the initial parallel version, but without compressing to a 1-dimensional array. Note how the scalable version needs fewer processors to solve large problems, and that it can solve arbitrarily large problems, while the other versions cannot go beyond a fixed problem size no matter how many processors are available. This is due to the imperfect load balancing in the earlier versions which were unable to allocate less than a single `s3` loop per processor.

Besides being able to compare alternative parallelizations, we can also compare to the work of others. Using only 16 processors of an IBM SP2 we solved the 3-arm, n=200 problem. This is approximately 500,000 times harder than the problem called "impractical" in [9], and 2,000 times harder than that solved in [3] on a Cray 2. Our system is only about 22× a single processor Cray 2, and hence the primary advantage is our serial and parallel optimizations.

Note that our work applies much more broadly than adaptive designs for clinical and preclinical trials, though this in itself is an important application. The bandit model is widely used in areas such as operations research, artificial intelligence, and game theory. Further, our work generally applies to neighbor recurrences using stencils. This common class of recurrences includes many dynamic programming problems such as the generalized string matching used in some data mining and bioinformatics applications, and includes other evaluation techniques such as backward induction and path induction.

Acknowledgements

This research was partially supported by NSF grants DMS-9504980 and DMS-0072910. Parallel computing support was provided by the University of Michigan's Center for Parallel Computing.

References

[1] Armitage, P. (1985), "The search for optimality in clinical trials", *Int. Statist. Rev.* **53**, pp. 1–13.

[2] Berry, D.A. and Fristedt, B. (1985), *Bandit Problems: Sequential Allocation of Experiments*, Chapman and Hall.

[3] Berry, D.A. and Eick, S.G. (1995), "Adaptive assignment versus balanced randomization in clinical trials — a decision-analysis", *Stat. in Medicine* **14**, pp. 231–246.

[4] Hardwick, J., Oehmke, R. and Stout, Q.F. (1999), "A program for sequential allocation of three Bernoulli populations", *Comp. Stat. and Data Analysis* **31**, pp. 397–416.

[5] Hardwick, J., Oehmke, R. and Stout, Q.F. (2001), "Optimal adaptive designs for delayed response models: exponential case", in *MODA6: Model Oriented Data Analysis*, Mueller, W. and Hackl, P., eds., to appear.

[6] Hardwick, J. and Stout, Q.F. (1998), "Flexible algorithms for creating and analyzing adaptive sampling procedures", *New Developments and Applications in Experimental Design*, IMS Lec. Notes–Mono. Series **34**, pp. 91–105.

[7] Hardwick, J. and Stout, Q.F. (1999), "Using path induction to evaluate sequential allocation procedures", *SIAM J. Scientific Computing* **21**, pp. 67–87.

[8] Ibarra, O.H., Wang, H. and Jiang, T. (1993), "On efficient parallel algorithms for solving set recurrence equations", *J. Algorithms* **14**, pp. 244–257.

[9] Kulkarni, R. and Kulkarni, V. (1987), "Optimal Bayes procedures for selecting the better of two Bernoulli populations", *J. Stat. Planning and Inference* **15**, pp. 311–330.

[10] Lew, A. and Halverson Jr., A. (1993), "Dynamic programming, decision tables, and the Hawaii parallel computer", *Computers and Mathematics with Applications* **27**, pp. 121–127.

[11] Lewandowski, G., Condon, A. and Bach, E. (1996), "Asynchronous analysis of parallel dynamic programming algorithms", *IEEE Trans. Parallel and Distributed Systems* **7**, pp. 425–438.

[12] Lokuta, B. and Tchuente, M. (1988), “Dynamic programming on two dimensional systolic arrays”, *Information Processing Letters* **29**, pp. 97–104.

[13] Malinowski, K. and Sadecki, J. (1986), “Dynamic programming: a parallel implementation”, *Parallel Processing Techniques for Simulation*, pp. 161–170.

[14] Ranka, S. and Sahni, S. (1990), “String editing on a SIMD hypercube multicomputer”, *J. Parallel and Distributed Computing* **9**, pp. 411–418.

[15] Rytter, W. (1988), “On efficient parallel computations for some dynamic programming problems”, *Theoretical Computer Science* **59**, pp. 297–307.

[16] Tang, D. (1995), “An efficient parallel dynamic programming algorithm”, *Computers and Mathematics with Applications* **30**, pp. 65–74.

[17] Sastry, R. and Ranganathan, N. (1993), “A systolic array for approximate string matching”, *Proc. IEEE Int’l Conf. on Computer Design*, pp. 402–405.

[18] Simon, R. (1977), “Adaptive treatment assignment methods and clinical trials”, *Biometrics* **33**, pp. 743–744.

[19] Strate, S.A., Wainwright, R.L., Deaton, E., George, K.M., Bergel, H., and Hedrick, G. (1993), “Load balancing techniques for dynamic programming algorithms on hypercube multicomputers”, *Applied Computing: States of the Art and Practice*, pp. 562–569.

[20] Wang, Y.-G. (1991), “Sequential allocation in clinical trials”, *Comm. in Statistics: Theory and Methods* **20**, pp. 791–805.

[21] Wei, L.J. and Durham, S. (1978), “The randomized play the winner rule in medical trials”, *J. Amer. Stat. Assoc.* **73**, pp. 840–843.